AN OPEN FRAMEWORK FOR UNSTRUCTURED GRID GENERATION

William T. Jones*
NASA Langley Research Center
Hampton, VA 23681-2199
w.t.jones@larc.nasa.gov

ABSTRACT

An open framework for the development of software applications in the field of unstructured numerical grid generation is presented. The goal of the framework is to define an Application Programming Interface that allows developers of grid generation software to seamlessly integrate alternative algorithms into their respective products. The presentation centers on decoupling the grid generation task into separate, largely independent component processes. The component processes are implemented following the guidelines of a standard software interface definition and can therefore be interchanged without the need to modify algorithm specifics. The approach contained herein provides for support of both legacy and emerging technology and is demonstrated through the description of a new unstructured grid generation package under development at the NASA Langley Research Center.

INTRODUCTION

The field of unstructured numerical grid generation has gained widespread acceptance in recent years with the introduction of techniques and software systems for the rapid, highly automated production of discretizations for complex domains[1-4]. Though there are a number of productive tools and techniques available, the state of the art has yet to fully mature. New algorithms and techniques are regularly being introduced. As such, it is in the interest of an application developer to have available the ability to quickly incorporate and leverage these alternative technologies with minimum impact on a dependent application. Also of interest is the ability to combine algorithms from multiple sources, where applicable, thereby expanding the impact and advantages of each.

Traditionally, individual algorithm developers directly incorporate the requirements of their scheme into its resulting software implementation. While this is sufficient for basic execution, it severely limits the expandability and growth potential of the algorithm. An example of such a shortcoming is a fixed method of geometry access in a surface triangulation algorithm. By adopting a single description for the defining geometry, and embedding it within the implementation, the triangulation developer will fix, for all practical purposes, the form of geometry supported by the algorithm. A change to the method of geometry access requires knowledge of, and the ability to edit, the triangulation algorithm. The level of work required for such a change depends highly on the forethought and style of the initial incorporation. The approach advocated here, however, is to define a set of standard software interfaces for key elements of the unstructured grid generation process. In this example, the interface separates the specifics of the method for geometry access from those of the triangulation algorithm. As such, the triangulation algorithm is largely independent of the form of the underlying geometry. It accesses geometry solely through the interface and does so without regard for the method of implementation supporting the interface. Therefore, a given method of geometry access can be later modified, or replaced, as dictated by arising needs without directly impacting the triangulation algorithm itself.

This paper considers the construction of such a standardized software interface for use in unstructured grid generation products. The interface construction starts by analyzing the overall process of unstructured grid generation. The process is broken down into its constituent component sub-processes for which software interfaces can be defined. From each component, a rule set of common requirements and results is constructed. The rule set defines the types of requests and expected outcomes available from each identified sub-process. From this common rule set the definition of a general Application Programming Interface (API) for unstructured grid generation results.

---

*Computer Engineer, Data Analysis and Imaging Branch

We consider here only the operations relevant to the task of unstructured grid generation and strive to encapsulate disciplines to minimize interdependence. We also make the distinction between the software interface, dependent algorithms, and derivative applications. The interface will be the software link that provides loose coupling between the implementation of algorithms which are dependent on the operations defined by the interface. A derivative application is one which makes use of the dependent algorithms, and possibly the interface directly, to accomplish the grid generation task.

PROCESS DECOUPLING

The process of unstructured grid generation, when analyzed, can be broken into a series of three major constituent sub-process components. Each sub-process may be viewed as loosely coupled to other sub-processes in that, while dependent on the data provided by another sub-process, it is independent of the specific details required to produce that data (i.e. implementation). This characteristic is what defines the methods of a given sub-process and makes possible the generic interaction of the sub-processes through a standard interface definition. The sub-processes identified in the current work are described in detail below and include geometry access, grid metric specification, and the primary goal of meshing. Each of the sub-processes is required to interact through the respective standard interface definitions in order to take full advantage of the benefits of interchangeability and encapsulation.

Geometry

The process of unstructured grid generation centers about a geometry of interest. Information required from the geometry is, in general, either of a query nature or results from the evaluation of an entity. The geometry may be represented in a variety of forms from discrete points to higher order spline representations, such as Non-Uniform Rational B-Splines (NURBS) and higher level Boundary Representations (B-Rep) [5, 6] from Computer Aided Design (CAD) systems. In addition to the basic geometry definition, topological information is required to relate how the individual entities are connected to construct the domain of interest. While this topological data is provided by B-Reps, and consequently by most major CAD systems, it must be provided by other means for less rigorous descriptions. In any event, the topological data is essential to the process of unstructured grid generation in that it provides the intelligence that allows the process to be automated. It is the topology that provides the boundaries of a surface to be meshed and the connectivity of those surface regions that form the domain boundary.

The current work employs the Computational Analysis Programming Interface (CAPrI)[7,8] as the basis for geometry access. CAPrI is a CAD-vendor neutral API used to access computational solid geometry related information directly from the kernel of the originating CAD system. The CAPrI API offers a layer of abstraction from the specific methods of a given CAD kernel's API while ultimately utilizing the original system used to create the subject geometry. This level of abstraction is in direct alignment with the approach presented here in that the abstraction insulates an application from the nuances of various supported CAD kernels. The API provides the operations that are common across the supported systems and provides for interrogation, data tagging, and the creation of solid primitives. By using the CAPrI API, an application is seamlessly integrated into all of the supported CAD systems without the need for software modification. CAPrI operations are restricted to manifold solid geometry, such as that defined by most modern CAD systems, and as such provides a closed topological description of the domain of interest. CAPrI also provides a closed tessellation[9] of the subject part that may be used to ensure physical consistency of the model. Therefore, inherent in the design of CAPrI, all of the geometric and topological information required for intelligently automated unstructured grid generation is available to derivative applications.

The geometry interface of this work draws largely from the design of CAPrI. However, an additional layer of abstraction is used to encapsulate the geometry operations such that they may be replaced or enhanced as directed by future needs. One such change might be to support geometry-only definitions. A primary example of such a definition is that of legacy IGES data. This type of data would require combination with a separate description of the topology for use in automated grid generation and as such does not fit into the current design goals of the CAPrI API. The additional abstraction used here will provide the possibility of such future support if deemed necessary.

Grid Metrics

The primary grid metric considered in the current work is element size. For this information, it is assumed that the API should be implemented in a manner so as to query data specified by the user. It is further assumed

that the caller of the API method will supply a location in the Cartesian space for which element size information is to be provided. Element size will be in the form of local edge lengths corresponding to each of three principal directions. From this information, a general second-order tensor can be defined to create the appropriate mappings to account for anisotropic grid generation[10-13]. In keeping with the theme of encapsulation, the details of the user's specification scheme are irrelevant. This specification can be of the form of a background mesh, edge seeding, analytic functions, etc. By using the standard grid metric interface, dependent algorithms are not directly linked to the specification scheme. Dependent algorithms are also able to seamlessly take advantage of any scheme chosen at a later date as implemented in a derivative application.

Meshing

The meshing process is broken into three phases. The phases are viewed in a hierarchical fashion and are defined by the geometrical entities that are to be discretized. First, the meshing of edges (the one dimensional curve entities bounding the surfaces that define the domain's extent) is considered. The two dimensional surface regions that bound the domain are then meshed subject to the results of the edge discretization. Finally, with the discretization of edges and bounding surfaces complete, the meshing of the volume interior can begin.

The phases are kept as separate procedures to add flexibility to the method of execution. For example, edges, instead of being incorporated into the surface meshing procedure, may either be discretized all at once prior to any surface meshing in a batch utility, or may be processed on an "as needed" basis in an interactive package. The latter method may be desired if the user is to be allowed to mesh individual groups of surfaces as part of an iterative grid evaluation process. This ordering of operations allows for faster individual surface computation in that only the edges bounding the subject surface(s) must be computed prior to surface meshing. Also, due to the two-manifold nature of the topology, some edges may have been computed by prior operations on an adjacent surface included in the current group. This method, however, does require added logistical support to ensure agreement of the edge and surface with current grid metric specifications. By separating the edge and surface discretization methods, the decision is delegated to the application developer based on the specific needs of the package. A similar case can be made for separating the surface and volume phases.

The meshing process depends on data produced by both the geometry and grid metric components. Geometric data is required for point placement, connectivity, etc. Grid metrics are required to prescribe the desired edge lengths and overall element quality. In order for the meshing algorithm to be truly extensible, the implementation should leverage the capabilities of the other component processes through the prescribed interface. Though not required, it is in the interest of the derivative application developer that this limit on interaction be imposed. This practice removes the dependency of the meshing algorithm from the details of the method of producing the required data. By relying on a proprietary means for producing similar results, the algorithm developer dictates the method to be used by resulting applications and therefore adversely limits the applicability of the algorithm.

APPLICATION PROGRAMMING INTERFACE

By identifying the major component sub-processes of the grid generation procedure, we are now in a position to define the API to be used for their interaction. Simply put, the API defines a set of rules to govern the interaction of dependent algorithms. The rule set also applies to the interaction of the derivative applications with respect to dependent algorithms. The API also defines a common entry point to each method of a component sub-process. As such, all dependent algorithms and derivative applications call the same routine regardless of the underlying implementation. Each method of the API can therefore be viewed as an abstraction satisfying a request, subject to some fixed set of input requirements, to produce an expected result. Another view of the API is that of a generic implementation of the component sub-processes.

Consider the following scenario where a procedure, *Function A*, requires data from a given component sub-process. Here *Function A* may represent a dependent algorithm or a derivative application. Two valid procedures to produce the data are represented by *Algorithm 1* and *Algorithm 2*. The data is provided through the API with *Method B*. What follows are three different approaches for implementing the API *Method B* in an attempt to seamlessly integrate the capabilities to produce the data of either *Algorithm 1* or *Algorithm2* in an unaltered *Function A*. Each approach results in *Method B* satisfying the request of *Function A*.

At its most rudimentary level, *Method B* can be a direct implementation of a specific algorithm. This is the case represented by Figure 1. *Method B* is a direct implementation in that it directly includes the coding of the *Algorithm 1* while following the interface definition and adhering to the API naming convention. Therefore the single API entry point, *Method B*, can only produce data via *Algorithm 1*. In other words, while "legal", this form is limited in scope to the use of *Algorithm 1* to produce the data.

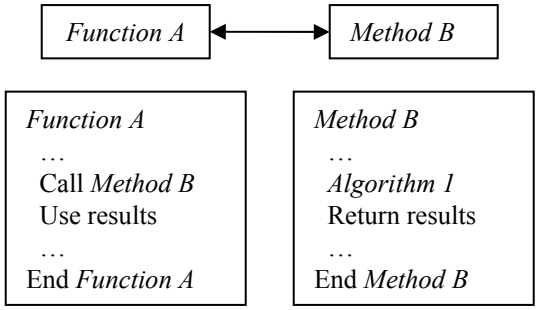


Figure 1 – Direct Implementation

Another approach would be to create *Method B* such that it acts as an intermediary. Figure 2 is a representation of this approach. Here *Method B* is aware of the "current algorithm" in use by the overall derivative application through the use of a global parameter. The parameter is then used as a switch to provide runtime selection of the desired algorithm. *Method B* can therefore make the appropriate call to produce the required data. This approach provides a great deal of flexibility to the application developer by allowing for user selection of the appropriate algorithm used to produce the desired data needed by *Function A* without the need to recode the derivative application or any of the algorithm implementations.

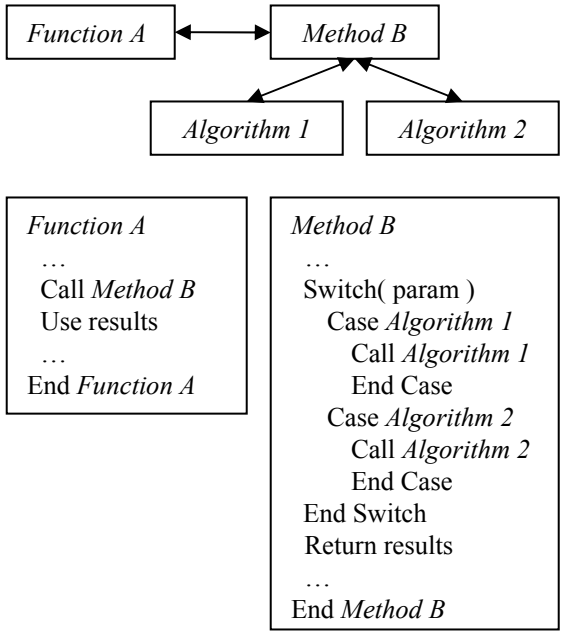


Figure 2 – Switched Implementation

Yet another approach might be to define multiple versions of the 1st approach, one for each implementation of a different algorithm used to produce the data. As shown in Figure 3, each implementation of *Method B* would then be used to construct a separate Dynamic Shared Object (DSO) library. Depending on the method of DSO selection at runtime, a capability to dynamically control the algorithm used to implement *Method B*, and therefore the operation of *Function A* can be achieved without requiring source code changes to the derivative application. This approach provides a means of adding capability to an existing application in keeping with the development of new technology.

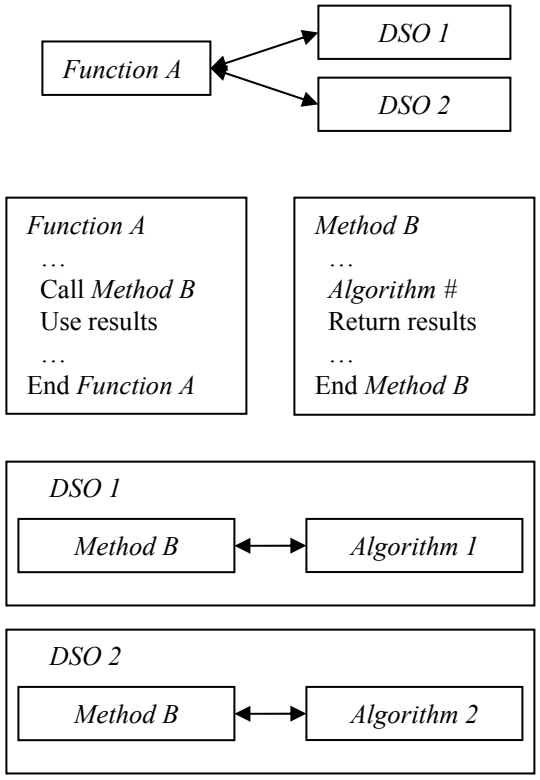


Figure 3 – DSO Implementation

In all approaches the API provides an abstraction layer to the algorithm implementations. This layer encapsulates modifications, enhancements, and errors within the algorithm implementation and prohibits the propagation of any changes into the dependent code.

Geometry

The geometry API used in the current work is a transparent abstraction of the CAPrI interface. Therefore, its description will not be duplicated here. Readers are referred to the CAPrI documentation[8] for additional information. However, as stated, the abstraction is used to allow for flexibility in future implementations. All interaction with the geometry will be conducted through the abstracted interface. This is in keeping with the general theme of encapsulation. Also the abstraction provides the ability to replace this pass thru implementation with any number of modifications or augmentations. Though not immediately foreseen, these changes may be required by future needs, and will as a result have minimal impact on any derivative code.

Grid Metrics

The API for grid metrics in the current work consists of a single method. The method is used to produce the desired cell edge lengths for a given location within the domain. A calling function makes a request of the method to return the edge lengths at a point in the Cartesian space of the subject domain. In return the method will satisfy the request with desired edge lengths relative to three given principal directions. The method from the current work has the following form in syntax similar to that of the C language.

```
GMetric_GetSpacing(x,y,z,s,dir)
  Double *x    - Target X coordinate
  Double *y    - Target Y coordinate
  Double *z    - Target Z coordinate
  Double s[3]  - Edge lengths
  Double dir[3]- Principal directions
```

The arguments of this method are all passed by reference to facilitate its use by dependent methods written in both the C and FORTRAN languages. The interface also defines this method as an integer function that returns a value indicative of the success of the operation.

Meshing

Each phase of the meshing process is assigned a separate interface within the API. Arguments to all methods are again passed by reference to facilitate their use by dependent methods written in both the C and FORTRAN languages. Again the interface also defines the methods as integer functions that return a value indicative of the success of the operation. Both the edge and surface meshing phases require the input of geometry entity identifiers. In keeping with the definitions given by the CAPrI API, the geometry identifiers will consist of both volume and entity ids. The volume id permits the meshing of entities from multiple domains as may be the case for an overset meshing application[14]. The entity id represents the target entity of the respective volume.

Edge meshing requires the additional input of the coordinates for the two bounding points, or nodes, on the defining curve. Upon satisfying the request, the edge meshing implementation will return the number of computed nodes, including the input bounding points, along with the physical and parametric coordinates of each node in the edge discretization. The parametric coordinates are relative to the defining curve of the target edge entity. The returned nodes are ordered monotonically from the first bounding point to the second, etc. This implies the orientation of the discrete edge segments. Coordinate values are returned as arrays with their respective components (i.e. x,y,z and u,v) as the major ordering. The current work defines the edge meshing interface as follows.

```
UGMesh_DiscretizeEdge(v,e,b,n,c,p)
  Integer  *v    - Target region
  Integer  *e    - Target edge entity
  Double   *b[3] - Bounding points
  Integer  *n    - Number of computed nodes
  Double   *c[3] - Output physical coords
  Double   *p[2] - Output parametric coords
```

The surface meshing interface requires geometry identifiers as input, but also requires information about the initial boundary discretization. The current interface only supports triangular surface meshing. The boundary information is constructed from data retrieved by prior use(s) of the edge meshing interface and assembled through access to the model topology using the geometry interface. The input data for the target surface consists of the number of initial fixed nodes, the fixed node physical coordinates, the number of boundary edge segments, and the boundary edge segment connectivity list. The boundary edge segment connectivity list is used to allow for multiple boundary loops and is oriented such that surface material on the left side of a counter-clockwise traversal of the segment data is maintained. The segment data references two

nodes per edge segment with the references relative to the fixed node inputs.

In addition to these required inputs, the interface supports the input of edge segments interior to the boundaries of the surface. These edge segments may be used to reference fixed nodes that are required of the resulting discretization, but are not part of the boundary definition. Such edge segments could be used to define non-manifold features on the interior of the surface such as corners, wires, pressure port taps, etc.

After satisfying a request, the face meshing implementation will return the number of computed nodes, including all fixed node inputs, along with the physical and parametric coordinates of each node in the surface discretization. Fixed nodes are listed first in the returned data. The parametric coordinates are relative to the target surface entity. As with the edge meshing interface, coordinate values are input and returned as arrays with their respective components as the major ordering. Also returned, are the number of computed elements and the computed element connectivity. The element connectivity is represented as three integer node references relative to the computed node list. This array has the node references as its major order.

The definition of the triangular surface discretization is listed below.

```
UGMesh_DiscretizeTriFace(v,s,nf,f,neb,eb,
                         nie,ie,n,c,p,ne,e)
  Integer *v     - Target region
  Integer *s     - Target surface entity
  Integer *nf    - Number of fixed nodes
  Double  *f[3]  - Fixed node coords
  Integer *neb   - Number of boundary segs
  Integer *eb[2] - Boundary segments
  Integer *nie   - Number of Interior segs
  Integer *ie[2] - Interior segments
  Integer *n     - Number of computed nodes
  Double  *c[3]  - Output physical coords
  Double  *p[2]  - Output parametric coords
  Double  *ne    - Num of computed elements
  Double  *e[3]  - Output element defs
```

Finally, the interface for the volume meshing implementation is defined. This interface is independent of geometry and is therefore only dependent on the input surface triangulations forming the boundary shell of the volume and the grid metric interface. The surface triangulations are computed with prior calls to the surface meshing interface and assembled using the underlying topology as accessed through the geometry interface. The volume meshing interface definition of the current work only supports the generation of tetrahedral volume elements.

Like the surface meshing interface, the volume meshing interface supports the input of additional non-manifold discretizations in the form of fixed nodes and connectivity. These non-manifold discretizations may be used by the underlying meshing implementation to define interior features. Both interior segments (wires, etc.) and interior collections of facets (sheets, wakes, etc.) are supported. The assembled boundary shell data is input to the volume meshing interface as: a number and list of fixed node coordinates; a number and list of the volume shell surface triangle element connectivity; a number and list of connectivity for interior facet elements; followed by the number and list of the interior edge segment connections.

Once the volume grid has been generated satisfying the initial request, the interface returns the data in the following form. The number of computed nodes and their respective physical coordinates are returned listing the fixed nodes at the beginning of this list in their original order. Also returned are the number and list of the volume element definitions. The definitions are represented by four integers that reference the computed node list and define the nodes used to create each element. This data assumes that the element definition is the major ordering.

Listed below is the volume discretization interface.

```
UGMesh_DiscretizeTetVol(v,nf,f,ns,se,nfi,
                        fi,nie,ie,n,c,ne,e)
  Integer *v     - Target region
  Integer *nf    - Number of fixed nodes
  Double  *f[3]  - Fixed node cords
  Integer *ns    - Number of shell elements
  Integer *se[3] - Shell elements
  Integer *nfi   - Number of interior faces
  Integer *fi[3] - Interior face triangles
  Integer *nie   - Number of interior segs
  Integer *ie[2] - Interior segments
  Integer *n     - Number of computed nodes
  Double  *c[3]  - Output physical coords
  Double  *ne    - Num of computed elements
  Double  *e[4]  - Output element definition
```

APPLICATION

The API presented above has been used in the creation of GridEx. This application is a product of the Fast Adaptive AeroSpace Tools (FAAST) element of the

Airframe Systems Concept to Test (ASCOT) program of the NASA Langley Research Center. One of the key components to the FAAST effort is the rapid generation and adaptation[15] of numerical grids directly from CAD models. The grid generation and adaptation capabilities are required to be independent of the originating CAD system, thereby providing support across the multitude of available systems. In addition, the capabilities must be implemented in a manner which facilitates the infusion of new techniques that result from FAAST research efforts. The GridEx application uses the framework approach presented in this work thereby providing the ability to incorporate multiple unstructured meshing techniques for tailored use by the FAAST team. The framework also facilitates the inclusion of emerging technological advances in unstructured methods over the life of the FAAST effort and beyond.

GridEx is an interactive tool used to construct baseline unstructured numerical grids for use in Computational Fluid Dynamics simulations. Within the tool, the user may interactively: define the domain(s) of interest surrounding the subject geometry; impose grid metric constraints to govern distribution of discrete grid points and the resulting element quality; individually or collectively generate surface grids for the constituent faces of the solid model; generate volume grids for the domain(s) of interest; and visualize the results. Each phase of the grid generation procedure is organized on a task oriented tabbed form located on a Graphical User Interface (GUI) as shown in Figure 4. The GUI also includes a 3D view of the problem space that can be manipulated interactively. A model tree is provided for the hierarchical organization of the problem to include boundary condition definition for output to the analysis software. Additional logistical functionality is provided by means of a standard application menu bar.

In addition to the framework basis of the GridEx application, one of the unique characteristics of the tool is the interaction between grid metric constraint specification and grid generation. The tool allows the user to specify grid metric constraints via the method of choice by setting appropriate parameters on the tabbed form. At any time during the specification, the user may elect to view the localized impact of the constraints on one or more selected surface meshes. The inspection requires the grid to be generated for the subject face(s) and any of the respective component edges. The grid generation is limited to those entities specified by the user and is thus very efficient. This process however may result in inconsistencies in the surface grid as faces fall out of sync with their neighbors. To allow for this iterative flexibility, a mechanism for automated consistency is built into the application via the framework. The method is summarized as follows. The framework maintains timestamps on the constraints and on each individual component grid (edge, face, and volume). As such the grids are aware of their state relative to the constraint specifications. Prior to volume grid generation all component grids are verified against the current state of the constraints. Component grids that are consistent with the constraints remain unchanged. Inconsistent component grids are automatically updated and

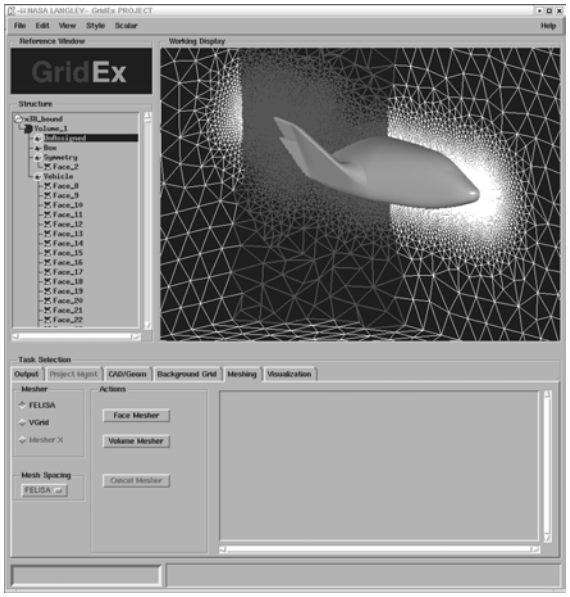


Figure 4 - GridEx Application

assembled for use in volume grid computation. This capability greatly reduces the time required to specify the desired metric constraints and provides flexibility and automatic consistency to the user.

GridEx provides the ability to define separate grid metric specifications and use them in conjunction with the available meshing algorithms to generate unstructured tetrahedral meshes directly from the CAD definition. The manifold solid model access provided by the CAPrI interface allows for automated topology extraction for use in the grid generation procedure. The user is responsible for specifying grid metric constraints prior to meshing and has the ability to iteratively generate grids on individual geometric entities while assessing the local impact of the constraints on the quality of the resulting mesh as stated above. Results from multiple metric/meshing algorithms can be compared within the same application session and is complemented by this iterative potential. Visualization

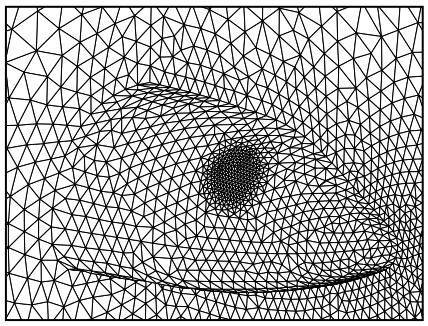
of grid data is provided in the interactive 3D window. Grid inspection is aided via flooded contour plotting of predefined grid quality measures. These contours can be viewed for any collection of boundary surfaces and/or "crinkle cuts" through the volume grid. The application is a testament to the applicability and benefits of the framework API.

GridEx currently allows the user to specify grid metric constraints using the algorithms found in FELISA[2] and VGRID[16]. These algorithms may be used seamlessly and interchangeably between both the FELISA and VGRID meshing implementations that are currently supported. Current implementations only support isotropic grid generation suitable for inviscid flow calculations, however work is in progress to support anisotropic stretching of elements to support viscous calculations. Both meshing implementations have been refactored into a modular set of API conforming libraries for use within the framework. The refactoring also included the use of the API to decouple the meshing algorithms from the underlying geometry and grid metric specification. As a result the framework now provides a matrix of capabilities to the GridEx user. The current matrix is populated as shown in Table 1 and is expected to be expanded in the future to support additional techniques as required by FAAST.

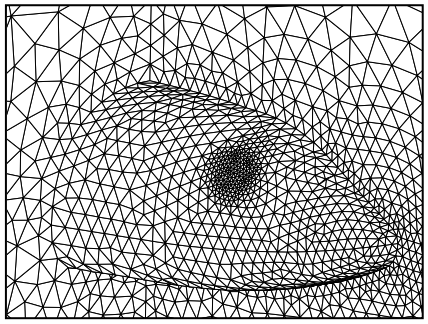
	FELISA Spacing	**VGRID Spacing**
FELISA Mesh	X	X
VGRID Mesh	X	X

Table 1 – Surface Grid Generation Capability

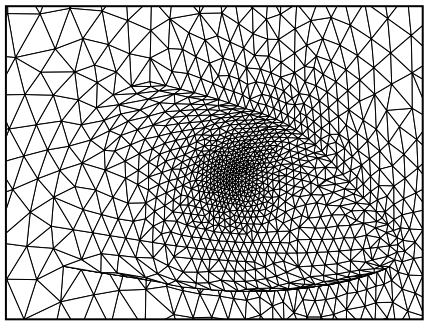
Examples of the flexibility afforded the user by the matrix of capabilities is demonstrated in Figure 5. The figure uses surface grids on the nose of a hypersonic vehicle to represent each of the four scenarios currently available for surface grid generation within GridEx. Figure 5a shows the result of a surface grid computed using the FELISA meshing algorithm and a traditional FELISA background grid. Figure 5b shows the same geometry meshed with the VGRID surface meshing algorithm and same FELISA background grid. The background grid used here consists of a line source that extends along the longitudinal axis of the vehicle and a single point source, centered in the clustered region of the figure, which was added for the purpose of demonstration. The point source was defined with a constant spacing distance 6 times that of the desired edge length defined for the source. The edge length doubling distance was specified as 10 times the source edge length. These parameters are detailed in the



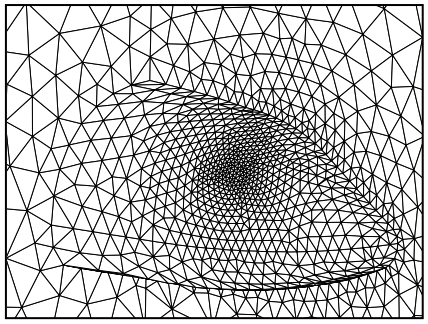
a) FELISA Mesh from FELISA Background Grid



b) VGRID Mesh from FELISA Background Grid



c) FELISA Mesh from VGRID Background Grid



d) VGRID Mesh from VGRID Background Grid

Figure 5 - Surface Grid Generation Capability

FELISA User's Guide[2]. Figures 5c and 5d show the same progression of meshing algorithm but with a VGRID background grid controlling grid clustering. Similarly, a line source is defined along the longitudinal axis and a point source is defined at the center of the clustered region. The effect of the point source with the VGRID background grid decays smoothly with increasing distance based on a user specified intensity value. Again, the details of the background grid definition are found in the literature[16].

When comparing surface grids generated with different meshing algorithms but the same background grid in Figure 5, only subtle differences are noted. This is reasonable as both grids adhere to the same metric constraints defined by the background grid. However, it is possible that other cases may yield more drastic differences. Fortunately, the decoupled framework provides the flexibility to choose the appropriate combination best suited for a particular problem at little or no cost. The reader is reminded that, as this figure demonstrates, future metric specification schemes can be added to the application with no impact to existing meshing algorithms.

A recent enhancement of the GridEx application comes from access to the viscous volume grid generation capabilities of the AFLR3 software. AFLR3 is a standalone grid generation package based on the advancing front local reconnection algorithm[3] and is capable of generating inviscid as well as viscous volume grids from an existing boundary triangulation. The tool has the ability to generate fully tetrahedral or mixed pentahedral boundary layer grids.

The framework basis of the GridEx application facilitated the integration of AFLR3 resulting in a total time to integrate of less than 12 hours. The integration method of choice was the switched implementation method detailed above. Modification to the framework was confined to the addition of a new switch branch within the volume API. However, special treatment was necessary as the AFLR3 package is invoked as a standalone executable code. As such, the "dependent algorithm" here consisted of a wrapper routine used to invoke the standalone executable. The purpose of the wrapper was to: create a disk file defining the boundary elements obtained through the API along with the associated boundary conditions; generate a script to control AFLR3 execution, also as a disk file; invoke a system call to execute the script; and finally import of the AFLR3 volume grid from the disk file generated by the execution. No refactoring of AFLR3 was possible for inclusion into the current work. As such, use of the API for dependent algorithms, namely grid metric constraint calculation, could not be accommodated. The definition of grid metric constraints for the volume grid is confined to that set by the AFLR3 application. The schemes available are based on interpolation, with various forms of decay, of the grid metrics prescribed by the supplied boundary triangulation.

Boundary triangulations for AFLR3 are generated using any of the available surface meshing techniques within GridEx and therefore are geometry conforming. In general, no new surface nodes are introduced as part of the volume grid generation. Element connectivity, however, may be altered as a result of local reconnection. Surface element connectivity is updated as part of the volume grid import process. However, viscous cell growth is allowed on planar symmetry surfaces. For these faces, new nodes and connectivity will be generated and both must be updated as part of the import process.

This inclusion of AFLR3 demonstrates the ability of the framework to handle proprietary and legacy code to the level of executing standalone packages while maintaining the cohesive nature of the API. The geometry depicted in Figure 6 was selected to demonstrate the capabilities afforded by the AFLR3 volume grid generator. Complexities included in this model are the struts used to connect the two vehicles as well as cavities along the wing trailing edge that represent gaps between flap surfaces. An example

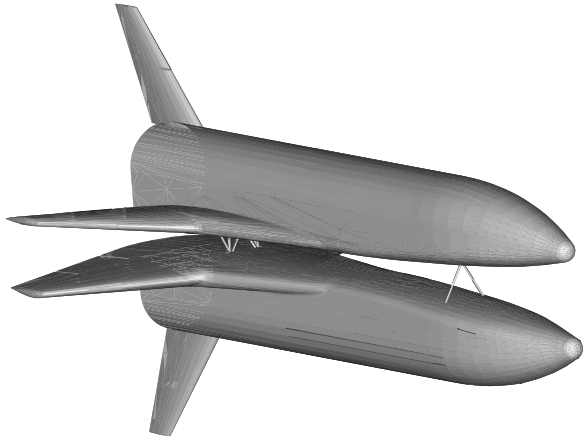


Figure 6 - Sample Geometry for use with AFLR3

viscous volume grid is shown in Figure 7. The insets show the smooth transition from the semi-structured viscous layers to the inviscid region of the grid. Not shown are the cavities representing the juncture of flap surfaces. Similar grid quality was obtained in these

regions. The boundary triangulation generated for this example where created using the FELISA surface grid generator and a FELISA background grid. The geometry was obtained in the form of a solid model from the Unigraphics commercial CAD system and processed using the Parasolid driver of CAPrI. The computational domain was defined by a Boolean subtraction of the geometry from a "Box" solid primitive created within GridEx and assumed half plane symmetry. This operation was carried out within GridEx. The resulting model was defined topologically with 412 edge and 134 face entities. This topological information was automatically extracted during the grid generation procedure.

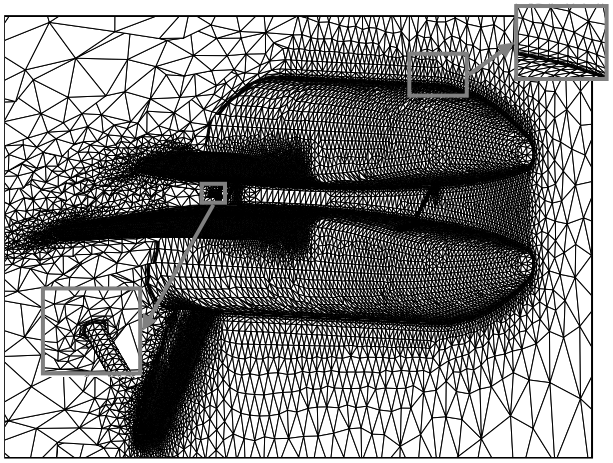


Figure 7 – Sample Viscous Grid from AFLR3

CONCLUSION

An example of an Application Programming Interface for the generation of unstructured grids for numerical analysis has been presented. Through the use of the interface, derivative applications as well as individual implementations of grid related algorithms can reduce their dependence on a single method of execution and thereby expand their potential for extension and future growth. Algorithms may be exchanged beneath the interface to provide alternative modes of operation and thereby expand their applicability. The API has been described and demonstrated with a sample application for the construction of unstructured tetrahedral volume grids.

ACKNOWLEDGEMENTS

The author wishes to thank the members of the CAD-Grid Working group from the FAAST team of the NASA Langley Research Center for their assistance and conversations regarding the current work. A former FAAST team member, K. James Weilmuenster (NASA LaRC ret.) was also a critical advocate of the current work. I also wish to extend thanks to Bob Haimes and Dr. Jaime Peraire of MIT for the exchange of ideas and generous support for their implementation. Appreciation is also extended to members of the Unstructured Grid Consortium for their acceptance of the framework outlined herein as a basis for a developing standard.

REFERENCES

[1]Parikh, P., Pirzadeh, S., and Löhner, R., "A Package for 3-D Unstructured Grid Generation," Finite Element Flow Solution and Flow Field Visualization, NASA CR-182090, 1990.

[2]Peiro, J., Peraire, J., and Morgan, K., "FELISA System Reference Manual and User's Guide, Volume 1," University of Wales Swansea Report, CR/821/94, 1994.

[3]Marcum, D. L., "Generation of Unstructured Grids for Viscous Flow Applications," AIAA Paper 95-0212, 1995.

[4]Pirzadeh, S., "Progress Towards A User-Oriented Unstructured Viscous Grid Generator," AIAA Paper 96-0031, 1996.

[5]Muuss, M. J., Butler, L. A., "Combinatorial Solid Geometry, Boundary Representations, and Non-Manifold Geometry," Stated of the Art in Computer Graphics: Visualization and Modeling, Rogers, D. F., Earnshaw, R. A., editors, Springer-Verlag, New York, pp. 185-223, 1991.

[6]Weiler, K., "The Radial Edge Structure: A Topological Representation for Non-Manifold Geometric Boundary Modeling," Geometric Modeling for CAD Applications, 1988.

[7]Haimes, R., "Computational Analysis Programming Interface," Proceedings of the 6th International Conference on Numerical Grid Generation in Computational Field Simulations, pp. 663-672, 1998.

[8]Haimes, R., "CAPRI: Computational Analysis PRogramming Interface User's Guide", Massachusetts Institute of Technology, 2001.

[9]Haimes, R., Aftomis, M., J., "On Generating High Quality Watertight Triangulations Directly from CAD," Proceedings of the 8th International Conference on Numerical Grid Generation in Computational Field Simulations, 2002.

[10]Castro-Díaz, M., J., Hecht, F., and Mohammadi, B., "New Progress in Anisotropic Grid Adaptation for Inviscid and Viscous Flows Simulations," Proceedings of the 4th International Meshing Roundtable, pp. 73-85, 1995.

[11]Bossen, F., J., and Hackbert, P., S., "A Pliant Method for Anisotropic Mesh Generation," Proceedings of the 5th International Meshing Roundtable, pp. 63-74, 1996.

[12]Borouchaki, H., Frey, P., J., and George, P., L., "Unstructured Triangular-Quadrilateral Mesh Generation. Application to Surface Meshing," Proceedings of the 5th International Meshing Roundtable, pp 229-242, 1996.

[13]Shimada, K., Yamada, A., and Itoh, T., "Anisotropic Triangular Meshing of Parametric Surfaces via Close Packing of Ellipsoidal Bubbles," Proceedings of the 6th International Meshing Roundtable, pp. 375-390, 1997.

[14]Nakahashi, K., Togashi, F., "Overset Unstructured Grid Method for Flow Simulation of Complex and Multiple Body Problems," ICAS Paper No. 0263, Presented at the 22nd International Congress of Aeronautical Sciences, Harrogate, United Kingdom, 2000.

[15]Park, M., A., "Adjoint-Based, Three-Dimensional Error Prediction and Grid Adaptation", AIAA Paper 2002-3286, 2002.

[16]Pirzadeh, S., "Structured Background Grids for Generation of Unstructured Grids by Advancing-Front Method", AIAA Journal 31:2, pp. 257-265, 1993.